Project 1: Optimizing the Performance of a Pipelined Processor

Zhuohao Li ✉[*], Xiaoheng Xia ✉[†]

April 24, 2022

1 Introduction

In this lab, we're gonna rewrite some code in Y86 assemble formats(Part A) and implement new instructions(Part B) by it. Besides, we need to modify the HCL description of the processor to add new instructions, optimizae the loop strategy and pipeline to reduce critical path, etc(Part C). It can be simulated and tested by running automated tests.

After this lab, we can master the operation of Y86 related assembly language and the description of Y86 HCL, and have a better understanding of pipeline and program optimization as well.

Zhuohao finished Part A,

Xiaoheng finished Part B,

Part C and this report.pdf are co-finished by both **Zhuohao** and **Xiaoheng**.

Note that we do not need the GUI interface throughout our experiments, thus it is a good idea to comment out the TK related lines in all Makefile(s).

2 Experiments

2.1 Part A

In this part, we are required to translate C code into Y86-64 assemble code. This part is not difficult.

2.1.1 Analysis

`sum.ys`

The first question requires writing a function to calculate the sum of sample linked lists. In `example.c int sum_list(list_ptr ls)`. We just follow the logic shown in C code. It should consist of some code that sets up the stack

[*]edith_lzh@sjtu.edu.cn | 519021911248

[†]xia.xh@sjtu.edu.cn | 519021910341

structure, invokes a function, and then halts. The most important thing is to figure out the instruction execution flow from C to Y86.

`rsum.ys`

This period is similar to `sum.ys` except change `int sum_list(list_ptr ls)` to `int rsum_list(list_ptr ls)`.

`copy.ys`

It should consist of code that sets up a stack frame, invokes a function copy block, and then halts. We need calculate the `XOR` value of all items in the original array and 3 parameters are passed in when the function is called here.

2.1.2 Code

`sum.ys`

```
# core code for sum_list(list_ptr ls)
sumlist:  pushl  %ebp   # push stack pointer
    rrmovl  %esp ,%ebp
    xorl   %eax,%eax     #the return val = 0, initialized
    mrmovl 8(%ebp) , %edx
    andl   %edx , %edx   #ls ==> %edx, to check if ls is 0/
        nullptr or not
    je  End       #if  ls == 0 go to End and return
Loop:  mrmovl (%edx) , %ecx    #ls->val ==> %ecx, in loop and
    store ls->value
    addl   %ecx , %eax   #val += ls->val, do adding
    irmovl $4 , %edi     #store 4
    addl   %edi , %edx   #next ==> edx, which is equivalent to
         ls+4
    mrmovl (%edx), %esi
    rrmovl %esi , %edx   #ls->next ==>edx
    andl   %edx , %edx   #check if ls is 0/nullptr
    jne Loop         #if ls != 0 go to Loop

End:   rrmovl  %ebp , %esp
    popl   %ebp
    ret
```

`rsum.ys`

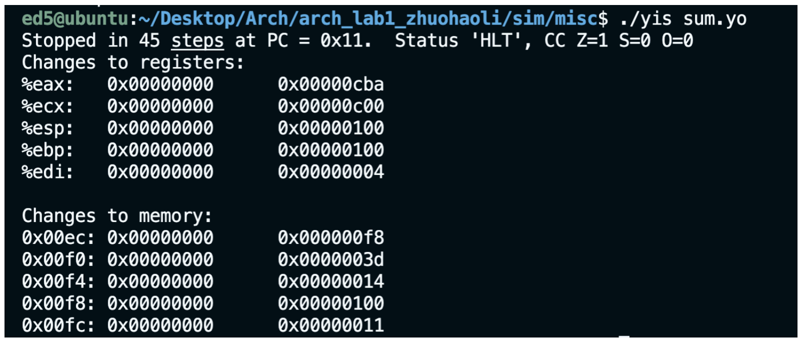
```
# core code for sum_rlist(list_ptr ls)
rsum_list:
    pushl  %ebp       # val is reserved as API register value
    rrmovl %esp , %ebp
    pushl  %ebx
    irmovl $4 , %esi
    subl   %esi , %esp
    xorl   %eax , %eax  #the return val = 0
    mrmovl 8(%ebp),%edx
    andl   %edx , %edx # condition set
    je  End      # if nullptr, go to End and return
    mrmovl (%edx) , %ebx #ls->val
    irmovl $4 , %esi
    addl   %esi , %edx
    mrmovl (%edx) , %edi
    rmmovl %edi , (%esp) #ls->next
    call   rsum_list    #recurit call
    addl   %ebx , %eax  #return val+rest_val

End:   addl   %esi , %esp
    popl   %ebx
    popl   %ebp
    ret
```

copy.ys

```
# core code for copy_block(int *src, int *dest, int len)
copy_block:
    pushl  %ebp
    rrmovl %esp, %ebp
    xorl   %eax , %eax #result = 0, initialized

    mrmovl 12(%ebp) , %edx #edx <==> dest
    mrmovl 8(%ebp) , %esi   #esi <==> src
    mrmovl 16(%ebp),%ecx    #ecx <==> len
    andl   %ecx, %ecx #length
    je  End    # if length <=0, go to End and return
Loop:  mrmovl  (%esi) , %ebx #val = *src
    rmmovl %ebx , (%edx)
    xorl   %ebx , %eax #result ^= val
    irmovl $4 , %edi
    addl   %edi , %edx #src++
    addl   %edi , %esi #dest++
    irmovl $1,%edi
    subl   %edi , %ecx #length--
    jne Loop
```

2.1.3 Evaluation

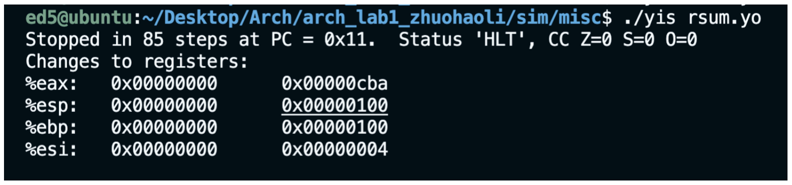
The evaluation is finished in `./misc`
`sum.ys` test by `./yas sum.ys;./yis sum.yo`, the result is shown in Fig1. which is correct.



```
ed5@ubuntu:~/Desktop/Arch/arch_lab1_zhuohaoli/sim/misc$ ./yis sum.yo
Stopped in 45 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000c00
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100
%edi:   0x00000000      0x00000004

Changes to memory:
0x00ec: 0x00000000      0x000000f8
0x00f0: 0x00000000      0x0000003d
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

Figure 1: `sum.ys` test

`rsum.ys` test by `./yas rsum.ys;./yis rsum.yo`, the result is shown in Fig2. They're the same.



```
ed5@ubuntu:~/Desktop/Arch/arch_lab1_zhuohaoli/sim/misc$ ./yis rsum.yo
Stopped in 85 steps at PC = 0x11.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x00000004
```

Figure 2: `rsum.ys` test

`copy.ys` test by `./yas copy.ys;./yis copy.yo`, the result is shown in Fig3. which is correct.

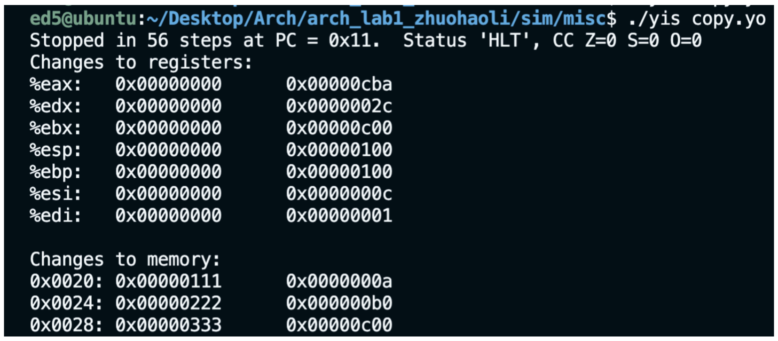
2.2 Part B

2.2.1 Analysis

In this project, we are going to implement an `iaddl` instruction. The function of `iaddl` is to add a constant value to a register. This requires first using an `irmovl` instruction to set a register to the constant, and then add `addl` instruction to add this value to the destination register.

According to the computing process of `irmovl` and `opl` from the book, we can write the computing process of the `iaddl`.

```
fetch:
    icode:ifun <-- M1[PC]
         rA:rB <-- M1[PC+1]
          valC <-- M4[PC+2]
          valP <-- PC+6
decode:
          valB <-- R[rB]
```



```
ed5@ubuntu:~/Desktop/Arch/arch_lab1_zhuohaoli/sim/misc$ ./yis copy.yo
Stopped in 56 steps at PC = 0x11.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x0000002c
%ebx:   0x00000000      0x00000c00
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000000c
%edi:   0x00000000      0x00000001

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
```

Figure 3: `copy.ys` test

```
execute:
        valE <-- valB + valC
# (There's no memory related operation.)
write back:
       R[rB] <-- valE
PC update:
          PC <-- valP
```

In the fetch phase, like most the other instructions, we read the instruction from `M1[PC]` and store it in `icode:ifun`. Then, we read the two operand: `rB` `,` `V` from `M1[PC+1]` and `M4[PC+2]` respectively. Lastly, we store `PC+6` in `valP`, as to provide a destination for the position of the next instruction.

In the decode phase, we only need to extract `R[rB]` and store it into `valB`, with `valC` already provided as immediate number.

In the execute phase, we use ALU to calculate `valB + valC` and store it into `valE`.

We don't need to read to or write from memory in this instruction.

In the write back phase, we write back the calculated result from `valE` to `R[rB]`

At last, we update Program Counter in the PC update phase.

So that we can modify the `seq-ful.hcl` file accordingly.

2.2.2 Code

Fetch stage: `IIADDL` should be added in the `instr_valid` section, and it needs register and immediate word `valC`.

```
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL };
```

```
# Does fetched instruction require a regid byte?
bool need_regids =
    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
             IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
```

```
# Does fetched instruction require a constant word?
bool need_valC =
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL
        };
```

Decode stage: Select register B as B source and E destination.

```
## What register should be used as the B source?
int srcB = [
    icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE; # Don't need register
];

## What register should be used as the E destination?
int dstE = [
    icode in { IRRMOVL } && Cnd : rB;
    icode in { IIRMOVL, IOPL, IIADDL } : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE; # Don't write any register
```

Execute stage:

As described before, we select `valC` and `valB` as ALU input. The default operation for ALU is `add` so we do not need to modify that. The condition code should be updated in this instruction.

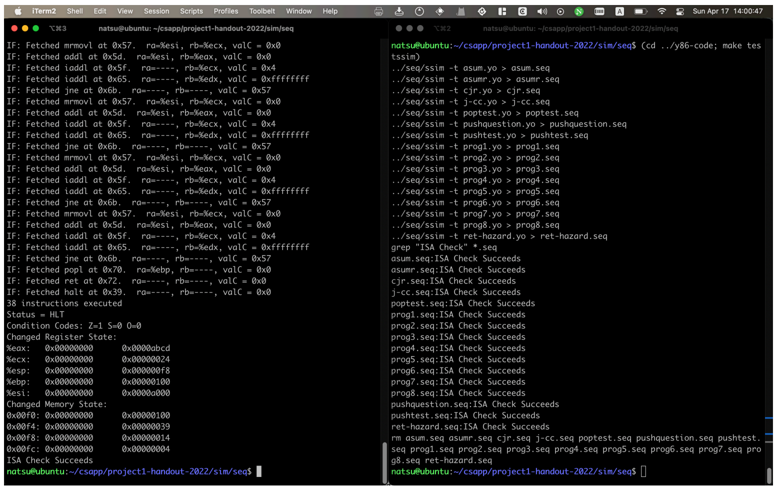
```
## Select input A to ALU
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
    icode in { ICALL, IPUSHL } : -4;
    icode in { IRET, IPOPL } : 4;
    # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
    icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
              IPUSHL, IRET, IPOPL, IIADDL } : valB;
    icode in { IRRMOVL, IIRMOVL } : 0;
    # Other instructions don't need ALU
];

## Should the condition codes be updated?
bool set_cc = icode in { IOPL, IIADDL };
```

Memory stage and PC update stage: `IADDL` operation do not involve in memory operations, and PC update stage do not need to be modified.

2.2.3 Evaluation

It is notable that we should also remove the GUI related code in the `makefile` to avoid compilation errors.

Testing your solution on a simple Y86 program:
`./ssim -t ../y86-code/asumi.yo`
Retesting your solution using the benchmark programs.
`(cd ../y86-code; make testssim)`
The results are shown in Figure 4.

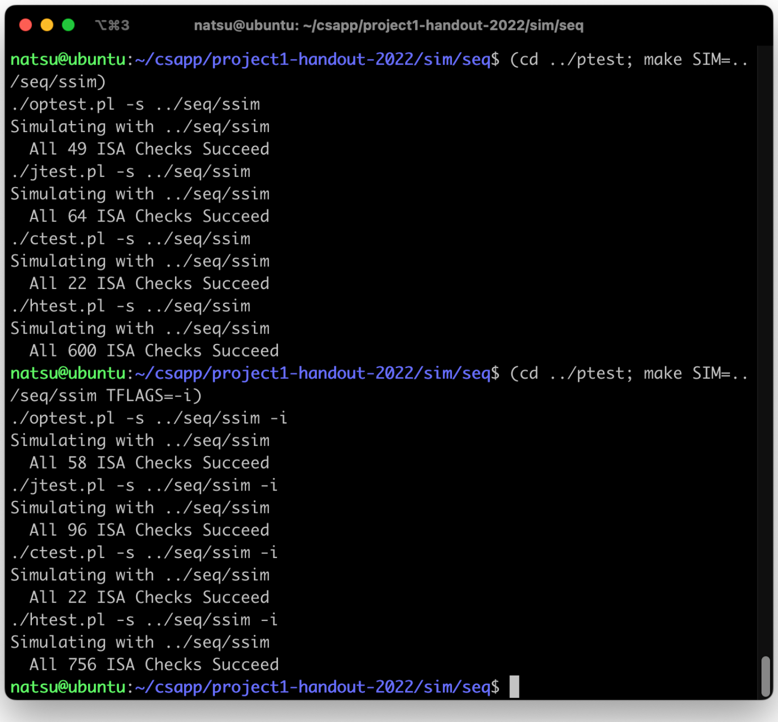

iTerm2
Shell
Edit
View
Session
Scripts
Profiles
Toolbelt
Window
Help
Sun Apr 17 14:00:47
natsu@ubuntu: ~/csapp/project1-handout-2022/sim/seq
natsu@ubuntu: ~/csapp/project1-handout-2022/sim/seq

```
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched popl at 0x70.  ra=%ebp, rb=----, valC = 0x0
IF: Fetched ret at 0x72.  ra=----, rb=----, valC = 0x0
IF: Fetched halt at 0x39.  ra=----, rb=----, valC = 0x0
38 instructions executed
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%esp:   0x00000000      0x000000f8
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000
Changed Memory State:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
ISA Check Succeeds
natsu@ubuntu:~/csapp/project1-handout-2022/sim/seq$
```

```
natsu@ubuntu:~/csapp/project1-handout-2022/sim/seq$ (cd ../y86-code; make tes
tssim)
../seq/ssim -t asum.yo > asum.seq
../seq/ssim -t asumr.yo > asumr.seq
../seq/ssim -t cjr.yo > cjr.seq
../seq/ssim -t j-cc.yo > j-cc.seq
../seq/ssim -t poptest.yo > poptest.seq
../seq/ssim -t pushquestion.yo > pushquestion.seq
../seq/ssim -t pushtest.yo > pushtest.seq
../seq/ssim -t prog1.yo > prog1.seq
../seq/ssim -t prog2.yo > prog2.seq
../seq/ssim -t prog3.yo > prog3.seq
../seq/ssim -t prog4.yo > prog4.seq
../seq/ssim -t prog5.yo > prog5.seq
../seq/ssim -t prog6.yo > prog6.seq
../seq/ssim -t prog7.yo > prog7.seq
../seq/ssim -t prog8.yo > prog8.seq
../seq/ssim -t ret-hazard.yo > ret-hazard.seq
grep "ISA Check" *.seq
asum.seq:ISA Check Succeeds
asumr.seq:ISA Check Succeeds
cjr.seq:ISA Check Succeeds
j-cc.seq:ISA Check Succeeds
poptest.seq:ISA Check Succeeds
prog1.seq:ISA Check Succeeds
prog2.seq:ISA Check Succeeds
prog3.seq:ISA Check Succeeds
prog4.seq:ISA Check Succeeds
prog5.seq:ISA Check Succeeds
prog6.seq:ISA Check Succeeds
prog7.seq:ISA Check Succeeds
prog8.seq:ISA Check Succeeds
pushquestion.seq:ISA Check Succeeds
pushtest.seq:ISA Check Succeeds
ret-hazard.seq:ISA Check Succeeds
rm asum.seq asumr.seq cjr.seq j-cc.seq poptest.seq pushquestion.seq pushtest.
seq prog1.seq prog2.seq prog3.seq prog4.seq prog5.seq prog6.seq prog7.seq pro
g8.seq ret-hazard.seq
natsu@ubuntu:~/csapp/project1-handout-2022/sim/seq$
```

Figure 4: Testing on [Left:] a simple Y86 program. [Right:] the benchmark program.

The results are all `ISA Check Succeeds`.
Performing regression tests: We test everything except `IADDL` using `(cd ../ptest; make SIM=../seq/ssim)` and we test the instruction `IADDL` using `(cd ../ptest; make SIM=../seq/ssim TFLAGS=-i)`.
The results are shown in Figure 5.
So that we've successfully completed the Part B lab.



```
natsu@ubuntu:~/csapp/project1-handout-2022/sim/seq$ (cd ../ptest; make SIM=..
/seq/ssim)
./optest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 49 ISA Checks Succeed
./jtest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 64 ISA Checks Succeed
./ctest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 600 ISA Checks Succeed
natsu@ubuntu:~/csapp/project1-handout-2022/sim/seq$ (cd ../ptest; make SIM=..
/seq/ssim TFLAGS=-i)
./optest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 58 ISA Checks Succeed
./jtest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 96 ISA Checks Succeed
./ctest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 756 ISA Checks Succeed
natsu@ubuntu:~/csapp/project1-handout-2022/sim/seq$
```

Figure 5: Regression tests

2.3 Part C

Part C is in the `sim/pipe`. We need modify `ncopy.ys` and `pipe-full.hcl` so that `ncopy.ys` runs as fast as possible.

2.3.1 Analysis

The `ncopy` function in Figure 2 copies a len-element integer array `src` to a non-overlapping `dst`, return- ing a count of the number of positive integers contained in `src`. There are a number of ways to optimize this function.

1. Implementing IIADDL

We're supposed to add `iaddl` in `pipe-full.hcl`, the implementation of the corresponding `iaddl` is similar to what we did in Part B. By simply substituting for the `iaddl` instruction, we can achieve better CPE.

2. Loop Unrolling

In `ncopy.ys`, we utilize **unrolling**, to cultivate 1 single loop to 4 loops (in the code, `LNpos#`).

It's basic idea is to unwind a loop process by rewriting each loop explicitly in the code. By trading code spaces for time, we can reduce branch penalties, thus optimize the program's execution speed.

Unrolling will increase the parallelism in the system and in VLSI and compiler design, we also utilize that kind of stuff to regularize and reduce our critical path and avoid stalls.

3. Bubble elimination

Additionally, there're **hazards** in original code like below:

```
mrmovl (%ebx), %esi # read val from src...
rmmovl %esi, (%ecx) # ...and store it to dst
```

Due to RAW hazards, there's a stall to avoid error. By carefully crafting the sequence of instructions, we add another `mrmovl` in the 2 instructions shown above to avoid unnecessary stalls. We can get more bonus from getting prepared value ahead, which is good for loop unrolling.

2.3.2 Code

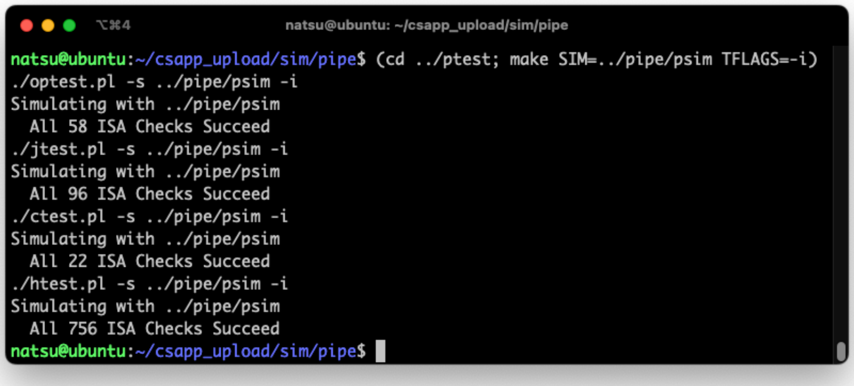
Though performing loop unrolling could make the code less readable, the code here is rather straightforward. Instead of copying one byte at a time, We copy 4 bytes at a time. If *len* < 4, then we do it 1 byte at a time, like the original code.

```
# You can modify this portion
# Loop Header
    xorl   %eax , %eax # count = 0;
    iaddl  $-4 , %edx #len = len -4
    andl   %edx , %edx # len <= 0?
    jl  remain
Loop:  mrmovl (%ebx) , %esi # read val from src...
    mrmovl 4(%ebx),%edi
    rmmovl %esi , (%ecx) # ...and store it to dst
    andl   %esi ,%esi # val <= 0?
    jle LNpos1 # if so, goto LNpos1:
    iaddl  $1 , %eax # count++ using iaddl
LNpos1: rmmovl %edi , 4(%ecx)
    andl   %edi , %edi
    jle    LNpos2
    iaddl  $1, %eax
LNpos2:mrmovl 8(%ebx) , %esi
    mrmovl 12(%ebx),%edi
    rmmovl %esi ,8 (%ecx)
    andl   %esi ,%esi
    jle LNpos3
    iaddl  $1 , %eax
LNpos3: rmmovl %edi , 12(%ecx)
    andl   %edi , %edi
    jle    nextLoop
    iaddl  $1, %eax
nextLoop:
    iaddl  $16,%ebx
    iaddl  $16,%ecx
    iaddl  $-4,%edx
    jge Loop

# maybe just remain less than 3
remain: iaddl  $4 , %edx # Restore the true len
    iaddl  $-1, %edx
    jl  Done
    mrmovl (%ebx) , %esi
    mrmovl 4(%ebx),%edi
    rmmovl %esi , (%ecx)
    andl   %esi ,%esi
    jle rNpos
    iaddl  $1 , %eax
rNpos:
    iaddl  $-1, %edx
    jl  Done
    rmmovl %edi , 4(%ecx)
    andl   %edi , %edi
    jle    rNpos1
    iaddl  $1, %eax
rNpos1:
    iaddl  $-1 , %edx
    jl  Done
    mrmovl 8(%ebx) , %esi
    rmmovl %esi , 8(%ecx)
    andl   %esi ,%esi
    jle Done
    iaddl  $1 , %eax
```

2.3.3 Evaluation

We add the instruction `IADDL` in `pipe-full.hcl` following the same process in Part B.

From Fig 6, we can see that we've successfully implemented `IADDL`.



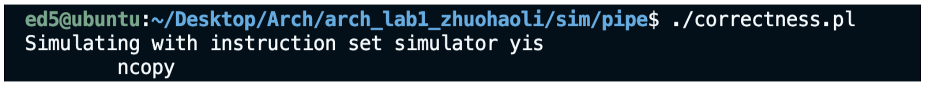
```
natsu@ubuntu:~/csapp_upload/sim/pipe$ (cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)
./optest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 58 ISA Checks Succeed
./jtest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 96 ISA Checks Succeed
./ctest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 22 ISA Checks Succeed
./htest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 756 ISA Checks Succeed
natsu@ubuntu:~/csapp_upload/sim/pipe$
```

Figure 6: IADDL implementation check

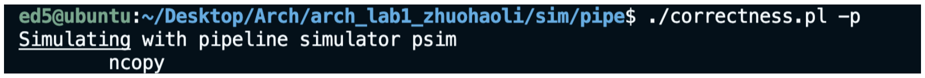
Following the guidance in assignment. We evaluate the correctness and efficiency.

Correctness

Using commands below to test the correctness.

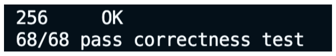


```
ed5@ubuntu:~/Desktop/Arch/arch_lab1_zhuohaoli/sim/pipe$ ./correctness.pl
Simulating with instruction set simulator yis
        ncopy
```



```
ed5@ubuntu:~/Desktop/Arch/arch_lab1_zhuohaoli/sim/pipe$ ./correctness.pl -p
Simulating with pipeline simulator psim
        ncopy
```

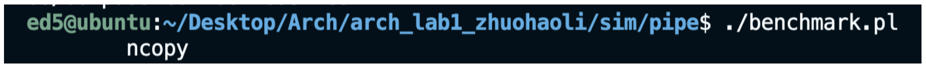
And the result is all CORRECT!(Fig 7)



```
256     OK
68/68 pass correctness test
```

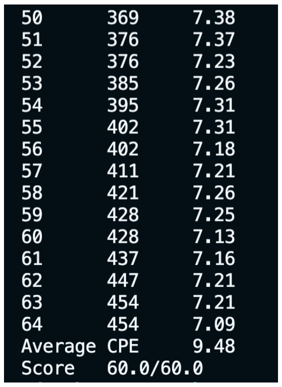
Figure 7: correctness check

efficiency Using `./benchmark.pl` in `./pipe`



```
ed5@ubuntu:~/Desktop/Arch/arch_lab1_zhuohaoli/sim/pipe$ ./benchmark.pl
        ncopy
```

And the Average CPE is **9.48**!(Fig. 8)

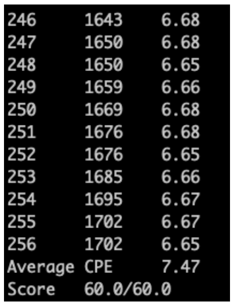


```
50      369     7.38
51      376     7.37
52      376     7.23
53      385     7.26
54      395     7.31
55      402     7.31
56      402     7.18
57      411     7.21
58      421     7.26
59      428     7.25
60      428     7.13
61      437     7.16
62      447     7.21
63      454     7.21
64      454     7.09
Average CPE     9.48
Score   60.0/60.0
```

Figure 8: speed upon benchmark

Indicating that we have finished this lab.

In addition, I've modified `benchmark.pl` to test our solution on a larger scale. We achieved a average CPE of 7.47 when the upper bound is set to 256, as shown in Fig 9



```
246     1643    6.68
247     1650    6.68
248     1650    6.65
249     1659    6.66
250     1669    6.68
251     1676    6.68
252     1676    6.65
253     1685    6.66
254     1695    6.67
255     1702    6.67
256     1702    6.65
Average CPE     7.47
Score   60.0/60.0
```

Figure 9: Larger scale benchmark

3 Conclusion

From this lab, we've familiarize ourselves with the Y86 instruction set. Now, we are able to write an assembly program from scratch, implement a new instruction by analyzing its computing process, and optimize a given assembly function by carefully avoiding hazards to achieve best CPE. Here is a summary of the completion of the three parts:

- Part A:
  - We familiarized ourselves with the Y86 instruction set.
  - We translated a C program into a Y86 assembly program.
- Part B:
  - We wrote out the computing process of the instruction `IADDL`
  - We modified the file `seq-full.hcl` to implement the instruction `IADDL`
- Part C:
  - We substitute for the `IADDL` instruction to speed up the process.
  - We did a 4-way loop unrolling to reduce branch penalties and optimize branch prediction.
  - We reordered the instructions to avoid RAW hazards and eliminate bubbles.

3.1 Feelings

This lab allows both of us to gain a deeper understanding in concepts and optimizations in Computer Architecture. By various hands-on approach, we debugged the problems and optimized the `ncopy` function step by step. We are grateful for this opportunity of being introduced to this valuable lab, and we would like to thank Prof. Shen and the TAs for their support.